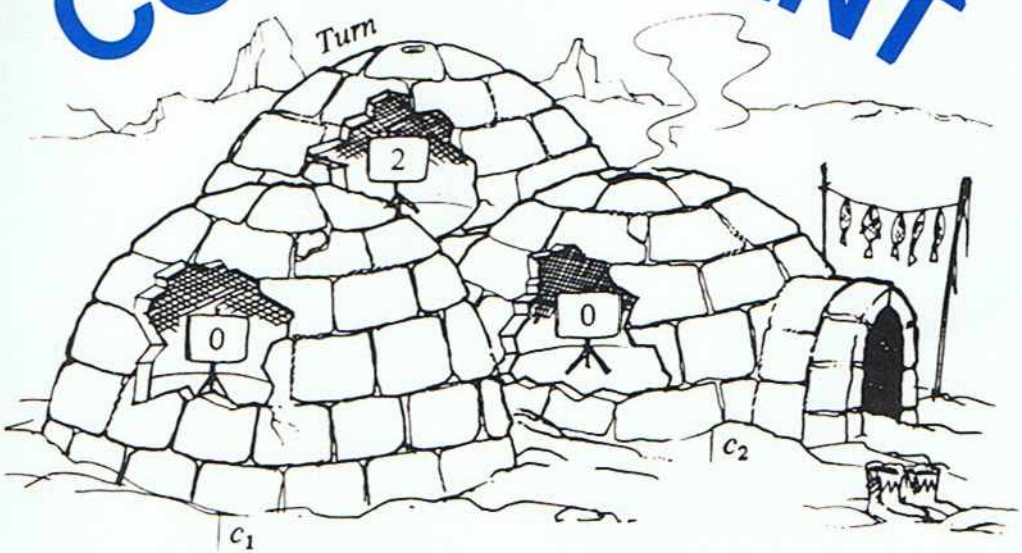
PRINCIPLES OF CONCURRENT PROGRAMMING



M. Ben-Ari

Principles of Concurrent Programming

M. Ben-Ari
Tel-Aviv University

Englewood Cliffs, New Jersey London New Delhi
Singapore Sydney Tokyo Toronto Wellington

1 WHAT IS CONCURRENT PROGRAMMING?

1.1 FROM SEQUENTIAL TO CONCURRENT PROGRAMMING

Figure 1.1 shows an interchange sort program. The program can be compiled into a set of machine language instructions and then executed on a computer. The program is sequential; for any given input (of 40 integers) the computer will always execute the same sequence of machine instructions.

If we suspect that there is a bug in the program then we can debug by tracing (listing the sequence of instructions executed) or by breakpoints and snapshots (suspending the execution of the program to list the values of the variables).

There are better sequential sorting algorithms (see Aho *et al.*, 1974) but we are going to improve the performance of this algorithm by exploiting the possibility of executing portions of the sort in parallel. Suppose that (for $n=10$) the input sequence is: 4, 2, 7, 6, 1, 8, 5, 0, 3, 9. Divide the array into two halves: 4, 2, 7, 6, 1 and 8, 5, 0, 3, 9; get two colleagues to sort the halves simultaneously: 1, 2, 4, 6, 7 and 0, 3, 5, 8, 9; and finally, with a brief inspection of the data, merge the two halves:

0
0, 1
0, 1, 2
. . .

A simple complexity analysis will now show that even without help of colleagues, the parallel algorithm can still be more efficient than the sequential algorithm. In the inner loop of an interchange sort, there are $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$ comparisons. This is approx. $n^2/2$. To sort $n/2$

```
program sortprogram;
const n=40;
var   a: array[1..n] of integer;
      k: integer;
procedure sort(low,high: integer);
var       i,j, temp: integer;
begin
   for i := low to high-1 do
            for j := i+1 to high do
            if a[j] < a[i] then
            begin
              temp := a[j];
              a[j] := a[i];
              a[i] := temp
            end
end;
begin (* main program *)
 for k := 1 to n do read (a[k]);
   sort (1, n);
   for k := 1 to n do write (a[k])
end.
```

Fig. 1.1.

elements, however, requires only $(n/2)^2/2 = n^2/8$ comparisons. Thus the parallel algorithm can perform the entire sort in twice $n^2/8 = n^2/4$ comparisons to sort the two halves plus another n comparisons to merge. The table in Fig. 1.2 demonstrates the superiority of the new algorithm. The last column shows that additional savings can be achieved if the two sorts are performed simultaneously.

n	$n^2/2$	$(n^2/4)+n$	$(n^2/8)+n$
40	800	440	140
100	5000	2600	1350
1000	500 000	251 000	126 000

Fig. 1.2.

Figure 1.3 is a sequential program for this algorithm. It can be executed on any computer with a Pascal compiler and it can be easily translated into other computer languages.

```
program sortprogram;
const n=20;
      twon=40;
var   a: array[1..twon] of integer;
      k: integer;
procedure sort(low, high: integer);
          (* as before *)
procedure merge(low, middle, high: integer);
var   count1, count2: integer;
      k, index1, index2: integer;
begin
  count1 := low;
  count2 := middle;
  while count1 < middle do
  if a[count1] < a[count2] then
    begin
      write (a[count1]);
      count1 := count1+1;
      if count1 >= middle then
      for index2 := count2 to high do
        write(a[index2])
    end
  else
    begin
      write (a[count2]);
      count2 := count2+1;
      if count2 > high then
        begin
          for index1 := count1 to middle-1 do
            write (a[index1]);
          count1 := middle (* terminate *)
        end
    end
end;
begin (* main program *)
  for k := 1 to twon do read (a[k]);
  sort(1, n);
  sort(n+1, twon);
  merge(1, n+1, twon)
end.
```

Fig. 1.3.

Suppose that the program is to be run on a multiprocessor computer—a computer with more than one CPU. Then we need some notation that can express the fact that the calls *sort*(1,*n*) and *sort*(*n*+1, *twon*) can be executed in parallel. Such a notation is the **cobegin–coend** bracket shown in Fig. 1.4. **cobegin** p_1; . . . ; p_n **coend** means: suspend the execution of the main program; initiate the execution of procedures $p_1, \ldots, p_n$ on multiple computers; when all of $p_1, \ldots, p_n$ have terminated then resume the main program.

```
program sortprogram;
( *declarations as before *)
begin (* main program *)
  for k := to twon do read(a[k]);
  cobegin
    sort(1, n);
    sort(n+1, twon)
  coend;
    merge(1, n+1, twon)
end.
```

Fig. 1.4.

The programs of Figs. 1.3 and 1.4 are identical except for the **cobegin–coend** in Fig. 1.4. There would be no need for both versions if the definition of **cobegin–coend** was modified. Instead of requiring that the procedures be executed in parallel, **cobegin–coend** becomes a declaration that the procedures *may* be executed in parallel. It is left to the implementation—the system hardware and software—to decide if parallel execution will be done. Processors may be added or removed from the system without affecting the correctness of the program—only the time that it would take to execute the program.

The word *concurrent* is used to describe processes that have the potential for parallel execution. We have shown how an algorithm can be improved by identifying procedures that may be executed concurrently. While the greatest improvement is obtained only under true parallel execution, it is possible to ignore this implementation detail without affecting the superiority of the concurrent algorithm over the sequential algorithm.

1.2 CONCURRENT PROGRAMMING

Concurrent programming is the name given to programming notations and techniques for expressing potential parallelism and for solving the resulting synchronization and communication problems. Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially

independent of concurrent programming. Concurrent programming is important because it provides an abstract setting in which to study parallelism without getting bogged down in the implementation details. This abstraction has proved to be so useful in writing clear, correct software that modern programming languages offer facilities for concurrent programming.

The basic problem in writing a concurrent program is to identify which activities may be done concurrently. If the *merge* procedure is also included in the **cobegin–coend** bracket (Fig. 1.5), the program is no longer correct. If you merge the data in parallel with sorting done by your two colleagues, the scenario of Fig. 1.6 might occur.

```
cobegin
  sort(1, n);
  sort(n+1, twon);
  merge(1, n+1, twon)
coend
```

Fig. 1.5.

	Colleague1	Colleague2	You
Initially	4, 2, 7, 6, 1	8, 5, 0, 3, 9	–
Colleague1 exchanges	2, 4, 7, 6, 1	8, 5, 0, 3, 9	–
Colleague2 exchanges	2, 4, 7, 6, 1	5, 8, 0, 3, 9	–
You merge	,,	,,	2
You merge	,,	,,	2, 4
You merge	,,	,,	2, 4, 5

Fig. 1.6.

However, *merge* could be a concurrent process if there were some way of synchronizing its execution with the execution of the sort processes (Fig. 1.7).

```
while count1 < middle do
  wait until i of procedure call sort(1,n) is greater than count1 and i of
  procedure call sort(n+1, twon) is greater than count2 and only then:
    if a[count1] < a[count2] then
    . . .
```

Fig. 1.7.

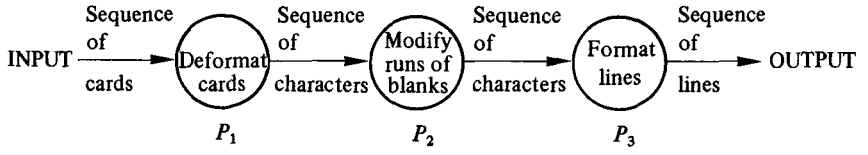
Parallelism is important not only for the improvement that can be achieved in program performance but also in program quality. Consider the

following problem:

> Read 80-column cards and print them on 125-character lines. However, every run of n = 1 to 9 blank spaces is to be replaced by a single blank followed by the numeral n.

This program is difficult to write as a sequential program. There are many interacting special cases: a run of blanks overlapping the end of a card, the pair blank-n overlapping the end of a line and so on. One way to improve the clarity of the program would be to write three separate programs: one to read cards and write a stream of characters onto a temporary file; a second program to read this character stream and modify runs of blanks, writing the new stream onto a second temporary file; and a third program to read the second temporary file and print lines of 125 characters each.

This solution is not acceptable because of the high overhead of the temporary files. However, if the three programs could be run concurrently (not necessarily in parallel) and communications paths could be established between them, then the programs would be both efficient and elegant.



Processes P_1, P_2, P_3 may execute concurrently

1.3 CORRECTNESS OF CONCURRENT PROGRAMS

Concurrent programming is much more difficult than sequential programming because of the difficulty of ensuring that a concurrent program is correct. Consider the sequential sort programs of Figs. 1.1 and 1.3: if they were tested on several sets of input data then we would feel confident that they are correct. Guidelines for testing would be to include a sorted input, a reversed input, an input all of whose elements are identical and so on. A run-of-the-mill bug (such as an incorrect **for**-loop limit) would seldom escape detection.

The scenario in Fig. 1.6. illustrates that the concurrent program of Fig. 1.5 is incorrect. However, this program is not a sequential program and other scenarios exist. If the processors assigned to sort are sufficiently rapid then *merge* may always be working on sorted data. In that case, no amount of testing would detect any problem. One day (perhaps months after the program has been put into production) an improved component in the computer system causes the merge to speed up and then the program gives incorrect answers as demonstrated in 1.6. Of course the natural reaction is:

"This program worked yesterday so the new component must be at fault."

A *scenario* is a description of a possible execution sequence of a program and shows how a computer might "act out" a program. It is usually used to show that a program is incorrect: since the computer may execute the program in a manner that produces the wrong answer, the program cannot be correct.

Conversely, how can we show that the concurrent program in Fig. 1.4 is correct? It no longer makes sense to look for and test paths that can be execution sequences. At times, there may be two such sequences caused by parallel execution of the algorithm.

Sequential programming has a well-developed proof theory. Assertions are made about the state of the computer (i.e. the values of the variables and the program counter) before and after executing an instruction, and these are then combined into a logical proof. In concurrent programming, this method needs to be modified because the programs can interfere with each other.

The correctness assertions for procedures *sort* and *merge* of the previous sections are elementary to state and prove:

sort input assertion: *a* is an array of integers,
sort output assertion: *a* is "sorted", i.e. *a* now contains a permutation of the original elements and they are in ascending order,
merge input assertion: the two halves of *a* are "sorted",
merge output assertion: the elements of *a* have been written in ascending order.

The correctness of the program in Fig. 1.1. is immediate from the correctness of procedure *sort*. The correctness of 1.3 is easily obtained by *concatenating* the correctness proofs of *sort* and *merge*. The correctness of Fig. 1.4 needs a new technique. We have to be able to express the fact that the two instances of *sort* do not interfere with one another. The program in Fig. 1.5 is incorrect though the procedures comprising it are correct; unfortunately, they interact in a manner which makes the program incorrect. The program in 1.7 is correct but new ideas are needed to be able to reason about synchronization.

1.4 INTERLEAVING

Interleaving is a logical device that makes it possible to analyze the correctness of concurrent programs. Suppose that a concurrent program P consists of two processes P_1 and P_2. Then we say that P executes any one of the execution sequences that can be obtained by interleaving the execution sequences of the two processes. It is as if some supernatural being were to execute the instructions one at a time, each time flipping a coin to decide whether the next instruction will be from P_1 or P_2.

We claim that these execution sequences exhaust the possible behaviors of P. Consider any instructions I_1 and I_2 from P_1 and P_2, respectively. If I_1 and I_2 do not access the same memory cell or register then it certainly does not matter if I_1 is executed before I_2, after I_2 or even simultaneously with I_2 (if the hardware so allows). Suppose on the other hand that I_1 is "Store 1 into memory cell M" and that I_2 is "Store 2 into memory cell M". If I_1 and I_2 are executed simultaneously then the only reasonable assumption is that the result is consistent. That is, cell M will contain either 1 or 2 and the computer does not store another value (such as 3) of its own volition.

If this were not true then it would be impossible to reason about concurrent programs. The result of an individual instruction on any given data cannot depend upon the circumstances of its execution. Only the external behavior of the system may change—depending upon the interaction of the instructions through the common data. In fact, computer hardware is built so that the result of executing an individual instruction is consistent in the way just defined.

Thus, if the result of the simultaneous execution of I_1 and I_2 is 1 then this is the same as saying that I_1 occurred before I_2 in an interleaving and conversely if the result is 2.

Interleaving does not make the analysis of concurrent programs simple. The number of possible execution sequences can be astronomical. Nevertheless, interleaved execution sequences are amenable to formal methods and will allow us to demonstrate the correctness of concurrent programs.

1.5 THE ORIGIN OF OPERATING SYSTEMS

Concurrent programming, though generally applicable, grew out of problems associated with operating systems. This section outlines the development of such systems so that the background to the growth of concurrent programming can be appreciated.

It is not often that an obsolete technology reappears. Programmable pocket calculators have resurrected machine language programming: absolute addresses must be used for data and labels. On the other hand, the owner is not constrained to working during the hours that the computer center is open.

While the pocket calculator is a marvel of electronics, machine language programming directly on the computer is slow and difficult. In the 1950s, when computers were few and expensive, there was great concern over the waste caused by this method. If you signed up to sit at the computer console from 0130 to 0200 and you spent 25 minutes looking for a bug, this 25 minutes of computer idle time could not be recovered. Nor was your colleague who signed up for 0200–0230 likely to let you start another run at 0158.

If we analyze what is happening in the terms of the previous sections we

see that the manual procedures that must be performed—mounting tapes, setting up card decks, or changing places at the console—are disjoint from the actual computation and can be performed concurrently with the computer's processing.

The second generation of computers used a supervisor program to batch jobs. A professional computer operator sat at the console. Programmers prepared card decks which were concatenated into "batches" that were fed into the computer once an hour or so. The increase in throughput (a measure of the efficiency of a computer; it is the number of jobs—suitably weighted—that can be run in a given time period) was enormous—the jobs were run one after another with no lost minutes. The programmers, however, lost the ability to dynamically track the progress of their programs since they no longer sat at the computer console. In the event of an error in one job, the computer simply commenced execution of the next job in the batch, leaving the programmer to puzzle out what happened from core dumps. With a turnaround time (the amount of time that elapses between a job being submitted for execution and the results being printed) of hours or days, the task of programming became more difficult even though certain aspects were improved by high-level languages and program libraries.

Despite this improvement in throughput, systems designers had noticed another source of inefficiency not apparent to the human eye. Suppose that a computer can execute one million instructions per second and that it is connected to a card reader which can read 300 cards per minute (= one card in 1/5 second). Then from the time the *read* instruction is issued until the time the card has been read, 200 000 instructions could have been executed. A program to read a deck of cards and print the average of the numbers punched in the cards will spend over 99% of its time doing nothing even though 5 cards per second seems very fast.

The first solution to this problem was *spooling*. The I/O speed of a magnetic tape is much greater than that of the card reader and the line printer that are the interface between the computer and the programmer. We can decompose the operation of the computer into three processes: a process to read cards to tape; a process to execute the programs on the tape and write the results onto a second tape; and a process to print the information from the second tape. Since these processes are disjoint (except for the exchange of the tapes after processing a batch), the throughput can be greatly increased by running each process on a separate computer. Since very simple computers can be used to transfer information to and from the magnetic tape, the increase in cost is not very great compared to the savings achieved by more efficient use of the main computer.

Later generations of computer systems have attacked these problems by switching the computer among several computations whose programs and data are held simultaneously in memory. This is known as *multiprogramming*. While I/O is in progress for program P_1 the computer will execute

several thousand instructions of program P_2 and then return to process the data obtained for P_1. Similarly, while one programmer sitting at the terminal of a time-sharing system† is thinking, the computer will switch itself to execute the program requested by a second programmer. In fact, modern computer systems are so powerful that they can switch themselves among dozens or even hundreds of I/O devices and terminals. Even a minicomputer can deal with a dozen terminals.

The importance of the concept of interleaved computations mentioned in the previous section has its roots in these multiprogrammed systems. Rather than attempt to deal with the global behavior of the switched computer, we will consider the actual processor to be merely a means of interleaving the computations of several processors. Even though multiprocessor systems—systems with more than one computer working simultaneously—are becoming more common, the interleaved computation model is still appropriate.

The sophisticated software systems that are responsible for multiprogramming are called *operating systems*. The term operating system is often used to cover all manufacturer-provided software such as I/O programs and compilers and not just the software responsible for the multiprogramming.

While the original concern of operating system designers was to improve throughput, it soon turned out that the throughput was affected by numerous system "crashes" when the system stopped functioning as it was supposed to and extensive recovery and restart measures delayed execution of jobs. These defects in the operating systems were caused by our inadequate understanding of how to execute several programs simultaneously and new design and programming techniques are needed to prevent them.

1.6 OPERATING SYSTEMS AND CONCURRENT PROGRAMMING

If you could sense the operation of a computer that is switching itself every few milliseconds among dozens of tasks you would certainly agree that the computer seems to be performing these tasks simultaneously even though we know that the computer is interleaving the computations of the various tasks. I now argue that it is more than a useful fiction to assume that the computer is in fact performing its tasks concurrently. To see why this is so, let us consider task switching in greater detail. Most computers use interrupts

† A time-sharing system is a computer system that allows many programmers to work simultaneously at terminals. Each programmer may work under the illusion that the computer is working for him alone (though the computer may seem to be working slowly if too many terminals are connected).

for this purpose. A typical scenario for task switch by interrupts is as follows. Program P_1 makes a read request and then has its execution suspended. The CPU may now execute program P_2. When the read requested by P_1 has been completed, the I/O device will interrupt the execution of P_2 to allow the operating system to record the completion of the read. Now the execution of either P_1 or P_2 may be resumed.

The interrupts occur asynchronously during the execution of programs by the CPU. By this is meant that there is no way of predicting or coordinating the occurence of the interrupt with the execution of any arbitrary instruction by the CPU. For example, if the operator who is mounting a magnetic tape happens to sneeze, it may delay the "tape ready" signal by 8.254387 seconds. However, if he is "slow" with his handkerchief, the delay might be 8.254709 seconds. Insignificant as that difference may seem, it is sufficient for the CPU to execute dozens of instructions. Thus for all practical purposes it makes no sense to ask: "What is the program that the computer is executing?" The computer is executing any one of a vast number of execution sequences that may be obtained by arbitrarily interleaving the execution of the instructions of a number of computer programs and I/O device handlers.

This reasoning justifies the abstraction that an operating system consists of many processes executing concurrently. The use of the term process rather than program emphasizes the fact that we need not differentiate between ordinary programs and external devices such as terminals. They are all independent processes that may, however, need to communicate with each other.

The abstraction will try to ignore as many details of the actual application as possible. For example, we will study the producer–consumer problem which is an abstraction both of a program producing data for consumption by a printer and of a card reader producing data for consumption by a program. The synchronization and communication requirements are the same for both problems even though the details of programming an input routine are rather different from the details of an output routine. Even as new I/O devices are invented, the input and output routines can be designed within the framework of the general producer–consumer problem.

On the other hand, we assume that each process is a sequential process. It is always possible to refine the description of a system until it is given in terms of sequential processes.

The concurrent programming paradigm is applicable to a wide range of systems, not just to the large multiprogramming operating systems that gave rise to this viewpoint. Moreover, every computer (except perhaps a calculator or the simplest microcomputer) is executing progams that can be considered to be interleaved concurrent processes. Minicomputers are supplied with small multiprogramming systems. If not, they may embedded in

real-time systems‡ where they are expected to concurrently absorb and process dozens of different asynchronous external signals and operator commands. Finally, networks of interconnected computers are becoming common. In this case true parallel processing is occurring. Another term used is *distributed processing* to emphasize that the connected computers may be physically separated. While the abstract concurrency that models switched systems is now well understood, the behavior of distributed systems is an area of current research.

1.7 AN OVERVIEW OF THE BOOK

Within the overall context of writing correct software this book treats the single, but extremely important, technical point of synchronization and communication in concurrent programming. The problems are very subtle; ignoring the details can give rise to spectacular bugs. In Chapter 2 we shall define the concurrent programming abstraction and the arguments that justify each point in the definition. The abstraction is sufficiently general that it can be applied without difficulty to real systems. On the other hand it is sufficiently simple to allow a precise specification of both good and bad behavior of these programs.

Formal logics exist which can formulate specifications and prove properties of concurrent programs in this abstraction though we will limit ourselves to informal or at most semi-formal discussions. The fact that the discussion is informal must not be construed as meaning that the discussion is imprecise. A mathematical argument is considered to be precise even if it is not formalized in logic and set theory.

The basic concurrent programming problem is that of mutual exclusion. Several processes compete for the use of a certain resource such as a tape drive but the nature of the resource requires that only one process at a time actually accessed the resource. In other words, the use of the resource by one process excludes other processes from using the resource. Chapter 3 presents a series of attempts to solve this problem culminating in the solution known as Dekker's algorithm. The unsuccessful attempts will each point out a possible "bad" behavior of a concurrent program and will highlight the differences between concurrent and sequential programs.

Dekker's algorithm is itself too complex to serve as a model for more complex programs. Instead, synchronization primitives are introduced. Just as a disk file can be copied onto tape by a single control language command

‡ Whereas a time-sharing system gives the user the ability to use all the resources of a computer, the term real-time system is usually restricted to systems that are required to respond to specific pre-defined requests from a user or an external sensor. Examples would be air-traffic control systems and hospital monitoring systems.

or a file can be read by writing *read* in a high level language, so we can define programming language constructs for synchronization by their semantic definition—what they are supposed to do—and not by their implementation. We shall indicate in general terms how these primitives can be implemented but the details vary so much from system to system that to fully describe them would defeat our purpose of studying an abstraction. Hopefully, it should be possible for a "casual" systems programmer to write concurrent programs without knowing how the primitives are implemented. A model implementation is described in the Appendix.

Chapter 4 commences the study of high level primitives with E. W. Dijkstra's semaphore. The semaphore has proved extraordinarily successful as the basic synchronization primitive in terms of which all others can be defined. The semaphore has become the standard of comparison. It is sufficiently powerful that interesting problems have elegant solutions by semaphores and it is sufficiently elementary that it can be successfully studied by formal methods. The chapter is based on the producer–consumer problem mentioned above; the mutual exclusion problem can be trivially solved by semaphores.

Most operating systems have been based on monolithic monitors. A central executive, supervisor or kernel program is given sole authority over synchronization. Monitors, a generalization of this concept formalized by Hoare, are the subject of Chapter 5. The monitor is a powerful conceptual notion that aids in the development of well structured, reliable programs. The problem studied in this chapter is the problem of the readers and the writers. This is a variant of the mutual exclusion problem in which there are two classes of processes: writers which need exclusive access to a resource and readers which need not exclude one another (though as a class they must exclude all writers).

The advent of distributed systems has posed new problems for concurrent programming. C. A. R. Hoare has proposed a method of synchronization by communication (also known as synchronization by rendezvous) appropriate for this type of system. The designers of the Ada programming language have chosen to incorporate in the language a variant of Hoare's system. Anticipating the future importance of the Ada language, Chapter 6 studies the Ada rendezvous.

A classic problem in concurrent programming is that of the Dining Philosophers. Though the problem is of greater entertainment value than practical value, it is sufficiently difficult to afford a vehicle for the comparison of synchronization primitives and a standing challenge to proposers of new systems. Chapter 7 reviews the various primitives studied by examining solutions to the problem of the Dining Philosophers.

Programming cannot be learned without practice and concurrent programming is no exception. If you are fortunate enough to have easy access to

a minicomputer or to a sophisticated simulation program, there may be no difficulty in practicing these new concepts. If not, the Appendix describes in full detail an extremely simple simulator of concurrency that can be used for class exercise. In any case, the Appendix can serve as an introduction to implementation of concurrency.

The book ends with an annotated bibliography suggesting further study of concurrent programming.

1.8 PROGRAM NOTATION

The examples in the text will be written in a restricted subset of Pascal-S, which is itself a highly restricted subset of Pascal. This subset must of course be augmented by constructs for concurrent programming. It is intended that the examples be legible to any programmer with experience in Pascal, Ada, C, Algol, or PL/I.

The implementation kit in the Appendix describes an interpreter for this language that will execute the examples and that can be used to program the exercises. The language in the kit contains more Pascal language features than are used in the text of the book and thus users of the kit are assumed to be able to program in sequential Pascal. These extra features are necessary in order to use the kit to solve the exercises, although the exercises themselves could be programmed in other languages that provide facilities for concurrent programming.

The examples in the chapter on monitors are standard and can be adapted to the many systems that provide the monitor facility such as Concurrent Pascal, Pascal-Plus, or CSP/k. The examples in the chapter on the Ada rendezvous are executable in Ada.

We now present a sketch of the language that should be sufficient to enable programmers unfamiliar with Pascal to understand the examples.

1. Comments are inserted between (* and *).
2. The first line in a program should be

   program *name*;

3. Symbolic names for constants may be declared by the word **const** followed by the constant itself:

   const *twon*=40;

4. All variables in each procedure in the main program must be declared by the word **var** followed by the names of the variables and a type:

   var *i*, *j*, *temp*: *integer*;
   found: *boolean*;

The available types are : *integer*, *boolean* (with constants *true* and *false*) and arrays:

var *a*:**array**[*lowindex...highindex*] **of** *integer*;

5. Following the declaration of the variables, procedures and functions may be declared: **procedure** *name* (*formal parameters*); and **function** *name* (*formal parameters*): *returntype*; . The formal parameter definition has the same form as that of a variable list:

procedure *sort* (*low*,*high*: *integer*);
function *last*(*index*: *integer*): *boolean*;

6. The body of the main program or of procedures is a sequence of statements separated by semi-colons between **begin** and **end**. The main program body is terminated by a period and the procedure bodies by semi-colons. The usual rules on nested scopes apply.

7. The statements are:

assignment statement
if *boolean-expression* **then** *statement*
if *boolean-expression* **then** *statement* **else** *statement*
for *index-variable* := *lowindex* **to** *highindex* **do** *statement*
while *boolean-expression* **do** *statement*
repeat *sequence-of-statements* **until** *boolean-expression*

The syntactic difference between **while** and **repeat** is that **while** takes a single statement and **repeat** takes a sequence of statements (separated by semi-colons). The semantic difference is that the **while** tests before the loop is done and **repeat** tests afterwards. Thus **repeat** executes its loop at least once.

8. A sequence of statements may be substituted for "statement" in the above forms by enclosing the sequence of statements in the bracket **begin** ... **end** to form a single "compound" statement:

```
if a[j] < a[i] then
  begin
    temp := a[j];
    a[j] := a[i];
    a[i] := temp
  end
```

In detail this is read: **if** the boolean expression ($a[j] < a[i]$) has the value *true*, **then** execute the compound statement which is a sequence of three assignment statements. If the expression is *false*, then the (compound) statement is not executed and the execution continues with the next statement.

9. Assignment statements are written variable := expression. The variable may be a simple variable or an element of an array: *a*[*i*]. The type (*integer* or *boolean*) of the expression must match that of the variable. Integer expressions are composed of integer variables and constants using the operators: +, −, *, **div** (integer divide with truncation) and **mod**. Boolean expressions may be formed from relations between integer expressions: =, <> (not equal), <, >, <= (less than or equal) >= (greater than or equal). The boolean operators **and**, **or** and **not** may be used to form compound boolean expressions.

10. For those who know Pascal we list here the additional features that are defined in the language of the implementation kit some of which will be necessary if you plan to write any programs using the kit.

    (a) Type declarations. Since there are no scalar, subrange or record types, this is mostly useful for array types:
    type *sortarray* = **array**[1..*n*] **of** *integer*
    var *a*: *sortarray*;
    (b) Character constants and variables of type **char**.
    (c) Multidimensional arrays (arrays of arrays).
    (d) A parameter may be passed by reference rather than value by prefixing the formal parameter by **var**.
    (e) Recursive functions and procedures.
    (f) I/O may be performed only on the standard textfiles *input* and *output*. To ensure that you do not forget this restriction, the declaration of external files in the **program** card has been removed. *read*, *readln*, *write*, *writeln*, *eoln*, *eof* (all without a file parameter) function as in Pascal. Only the default field widths may be used in a *write*, which will, however, accept a 'string' as a field to be printed:
    writeln ('*the answer is*', *n*).

1.9 EXERCISES†

1.1 Write a two-process concurrent program to find the mean of *n* numbers.

1.2 Write a three-process concurrent program to multiply 3×3 matrices.

1.3 Each process of the matrix multiply program executes three multiplications and two additions for each of three rows or altogether 15 instructions. How many execution sequences of the concurrent program may be obtained by interleaving the executions of the three processes?

† Slightly harder exercises are marked throughout the book with an asterisk (*).

1.4 Perform a similar analysis for *sortprogram*. You will have to make some assumptions on the number of interchanges that will be done.

1.5 Test the concurrent *sortprogram* of Fig. 1.3.

1.6 Test the concurrent *sortprogram* of Fig. 1.4 which has the *merge* defined as a third process. Run the program several times with exactly the same data.

1.7 Run the program in Fig. 1.8 several times. Can you explain the results?

```
program increment;
const m = 20;
var n: integer;
procedure incr;
var i: integer;
begin
  for i := 1 to m do n := n+1
end;
begin (* main program *)
  n := 0;
  cobegin
    incr; incr
  coend;
  writeln (' the sum is ', n)
end.
```

Fig. 1.8.

BIBLIOGRAPHY

TEXTBOOKS

Concurrent programming is an abstraction of the type of programming that is common in operating systems, and in fact its study is rooted in the difficulties that were encountered in programming operating systems. Thus it is not surprising that other books on this subject deal with operating systems and conversely that books on operating systems usually have a short chapter on concurrent programming. Exceptions are the texts by Holt *et al.* (1978) and Brinch Hansen (1973). The text by Holt describes the construction and use of the CSP/k system for exercising concurrency. CSP/k uses monitors. The text contains a good description of monitor programming and of implementation of concurrency. Its treatment of other synchronization primitives is sketchy.

Brinch Hansen's text is addressed to about the same level as this book. His discussion emphasizes the conditional critical region.

Books on operating systems are those by Tsichritzis and Bernstein (1974) at an elementary level, by Brinch Hansen (1973) and Habermann (1976) at an intermediate level, and by Coffman and Denning (1973) at an advanced level.

One of my favorite books is Brinch Hansen (1977) which contains a description of the language Concurrent Pascal (which uses monitors) and the design and listing of three operating systems.

For an introduction to Pascal, the original reference manual is that by Jensen and Wirth (1975); there are numerous newer texts such as that by Welsh and Elder (1979). An introduction to preliminary Ada is provided by Wegner (1980). There is a newer text by Pyle (1981). For the design and construction of programs see Welsh and McKeag (1980).

SOURCES

The preliminary report on the Ada language is Ichbiah (1979). The manual of the revised language is available from the U.S. Superintendent of Documents, Washington, D.C. Most of the rest of the text is based on the excellent articles by Dijkstra (1968b) and Hoare (1974) both of which should be read. The dining philosophers are discussed in Dijkstra (1971) and Hoare and Perrott (1972).

Conway's problem is in Conway (1963) and appears again later in Hoare (1978). While Hoare's system CSP was the basis for the Ada synchronization primitives, CSP is different and worth learning. Most current theoretical work is being done on CSP rather than Ada: Francez (1980) and Apt, Francez and de Roever (1980). Guarded commands were invented by Dijkstra (1975). Another formalism for distributed synchronization has been published by Brinch Hansen (1978).

Lamport's bakery algorithms are described and proved in his papers of 1974, 1977 and 1979. The exercise attributed to Roussel is found in Kowalski (1979). The appendix is based on a program that the Author has used for class exercise for several years. A short description of a preliminary version was published in Ben-Ari (1981). The Pascal-S report has been reprinted in Barron (1981). More sophisticated systems are described in Holt *et al.* (1978) and Kaubisch *et al.* (1976). If you have access to *Concurrent Pascal* (Brinch Hansen, 1977), that is preferable.

The proofs of the programs in the text are semi-formal expositions of formal proofs I have been working on using temporal logic. If this interests you, places to start are Manna and Pnueli (1979) for sequential programs and Pnueli (1981) for concurrent programs. Compare the proof of Dekker's Algorithm given in this book with that in the article by Francez and Pnueli (1978). For another approach to proving concurrent programs see Owicki and Gries (1976).

To test your knowledge of concurrent programming, make the effort needed to understand the mutual exclusion algorithm in Morris (1979).

Aho, A. V., J. E. Hopcroft and J. D. Ullmann, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, Mass. (1974).

Apt, K., N. Francez and W. P. de Roever, "A proof system for communicating sequential processes", *ACM Trans. on Programming Langauges and Systems*, **2**, 359–385 (1980).

Barron, D. W., *Pascal—The Language and its Implementation*, John Wiley, Chichester (1981).

Ben-Ari, M., "Cheap concurrent programming", *Software—Practice and Experience*, **11**, 1261–1264 (1981).

Brinch Hansen, P., *Operating Systems Principles*, Prentice-Hall, Englewood Cliffs, N.J. (1973).

Brinch Hansen, P., *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood Cliffs, N.J. (1977).

Brinch Hansen, P., "Distributed processes: a concurrent programming concept", *Comm. ACM*, **21**, 934–941 (1978).

Coffman, E. G., Jr. and P. J. Denning. *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, N.J. (1973).

Conway, M. E., "Design of a separable transition—diagram compiler", *Comm. ACM*, **6**, 396–408 (1963).

Courtois, P. J., F. Heymans and D. L. Parnas, "Concurrent control with readers and writers", *Comm. ACM*, **14**, 667–668 (1971).

Dijkstra, E. W., "The structure of the T.H.E. multiprogramming system", *Comm. ACM*, **11**, 341–346 (1968a).

Dijkstra, E. W., "Cooperating sequential processes" in F. Genuys (ed.) *Programming Languages*, Academic Press, New York (1968b).

Dijkstra, E. W., "Hierarchical ordering of sequential processes", *Acta Informatica*, **1**, 115–138 (1971).

Dijkstra, E. W., "Guarded commands, nondeterminacy and formal derivation of programs," *Comm. ACM*, **18**, 453–457 (1975).

Francez, N., "Distributed termination", *ACM Trans. on Programming Languages and Systems*, **2**, 42–55 (1980).

Francez, N. and A. Pnueli, "A proof method for cyclic programs", *Acta Informatica*, **9**, 133–157 (1978).

Habermann, A. N., *Introduction to Operating System Design*, Science Research Associates, Chicago (1976).

Hoare, C. A. R. and R. H. Perrott (eds.), *Operating Systems Techniques*, Academic Press, New York (1972).

Hoare, C. A. R., "Monitors: an operating system structuring concept", *Comm. ACM*, **17**, 549–557 (1974).

Hoare, C. A. R., "Communicating sequential processes", *Comm. ACM*, **21**, 666–677 (1978).

Holt, R. C., G. S. Graham, E. D. Ladzowska and M. A. Scott, *Structured Concurrent Programming with Operating Systems Applications*, Addison–Wesley, Reading, Mass. (1978).

Howard, J. H., "Proving monitors", *Comm. ACM*, **19**, 273–279 (1976).

Ichbiah J., "Preliminary Ada Reference Manual and Rationale for the Design of the Ada Programming Language", *SIGPLAN Notices*, **14**(6) (1979).

Jensen, K. and N. Wirth, *Pascal User Manual and Report*, *Springer-Verlag Lecture Notes in Computer Science*, **18**, Berlin (1975).

Kaubisch, W. H., R. H. Perrott and C. A. R. Hoare, "Quasiparallel programming", *Software—Practice and Experience*, **6**, 341–356 (1976).

Kessels, J. L. W. and A. J. Martin, "Two implementations of the conditional critical region using a split binary semaphore", *Information Process. Lett.*, **8**(2), 67–71 (1979).

Kowalski, R., "Algorithm = Logic + Control", *Comm. ACM*, **22**, 424–436 (1979).

Lamport, L., "A new solution of Dijkstra's concurrent programming problem", *Comm. ACM*, **17**, 453–455 (1974).

Lamport, L., "Proving the correctness of multiprocess programs", *IEEE Trans. Software Eng.*, **SE–3**, 125–143 (1977).

Lamport, L., "A new approach to proving the correctness of multiprocess programs", *ACM Trans. on Programming Languages and Systems*, **1**, 84–97 (1979).

Manna, Z. and A. Pnueli, "The modal logic of programs: a temporal approach", *Automata, Languages and Programming*, *Springer-Verlag Lecture Notes in Computer Science*, **79**, 385–409 (1979).

Morris, J. M., "A starvation-free solution to the mutual exclusion problem", *Information Process. Lett.*, **8**, 76–80 (1979).

Owicki, S. and D. Gries, "Verifying properties of parallel programs: an axiomatic approach", *Comm. ACM*, **19**, 279–285 (1976).

Parnas, D. L., "On a solution to the cigarette smoker's problem (without conditional statements)", *Comm. ACM*, **18**, 181–183 (1975).

Pnueli, A., "The temporal semantics of concurrent programs", *Theoret. Computer Sci.*, **13**, 45–60 (1981).

Pyle, I. C., *The Ada Programming Language*, Prentice-Hall International, London (1981).

Tsichritzis, D. C. and P. A. Bernstein, *Operating Systems*, (*Computer Science and Applied Mathematics Series*), Academic Press, New York (1974).

Wegner, P., *Programming with Ada*, Prentice-Hall, Englewood Cliffs, N.J. (1980).

Welsh, J. and J. Elder, *Introduction to Pascal*, Prentice-Hall International, London (1979).

Welsh, J. and M. McKeag, *Structured System Programming*, Prentice-Hall International, London. (1980).